WEB TECHNOLOGIES

NodeJS & MongoDB

Prof. Spurthi N Anjan
Department of Computer Science and Engineering
spurthianjan@pes.edu

Backend Development

Almost every application is incomplete without a functioning server and database to process data. Now that we can easily create front ends using React, let us understand why leading companies like Netflix, Uber, and LinkedIn use Node.js as their the backend development platform

What is NodeJS?

Node.js is an open-source server side runtime environment built on Chrome's V8 JavaScript engine.

Simply put, **Node.js** is a platform that executes server-side JavaScript programs that can communicate with I/O sources like networks and file systems.

It is essentially JavaScript outside of a browser.

It' is designed to build scalable network applications

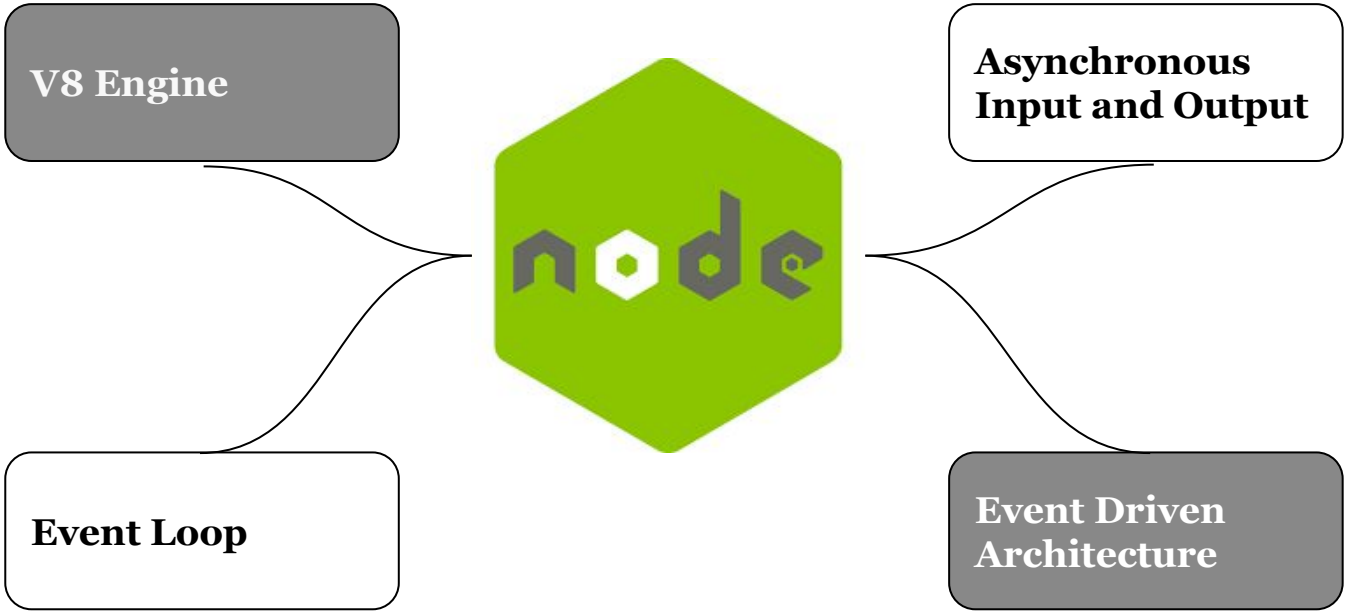
Why NodeJS?

When Ryan Dahl created Node in 2009 he argued that I/O was being handled incorrectly, blocking the entire process due to synchronous programming.

Traditional web-serving techniques use the thread model, meaning one thread for each request. Since in an I/O operation the request spends most of the time waiting for it to complete, intensive I/O scenarios entail a large amount of unused resources (such as memory) linked to these threads. Therefore the “one thread per request” model for a server doesn’t scale well.

Instead of the thread model, he said the right way to handle several concurrent connections was to have a single-thread, an event loop and non-blocking I/Os.

The Essence of NodeJS

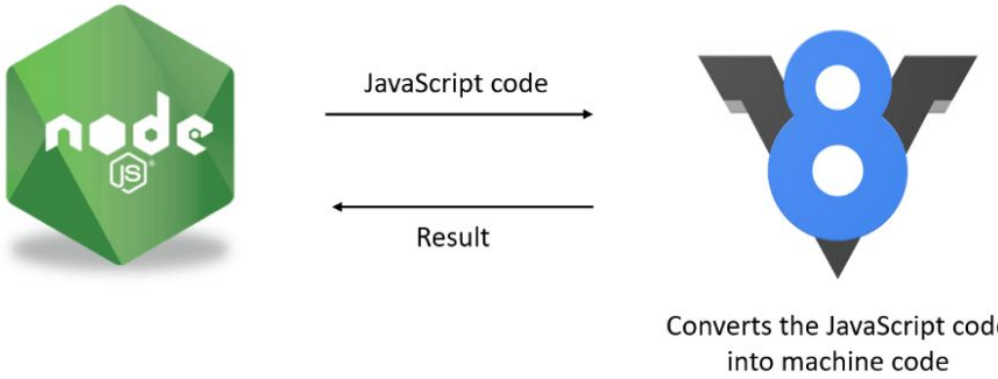


V8 Engine

Node.js is built on the V8 engine of Google. Node.js cannot understand the javascript code we write without V8.

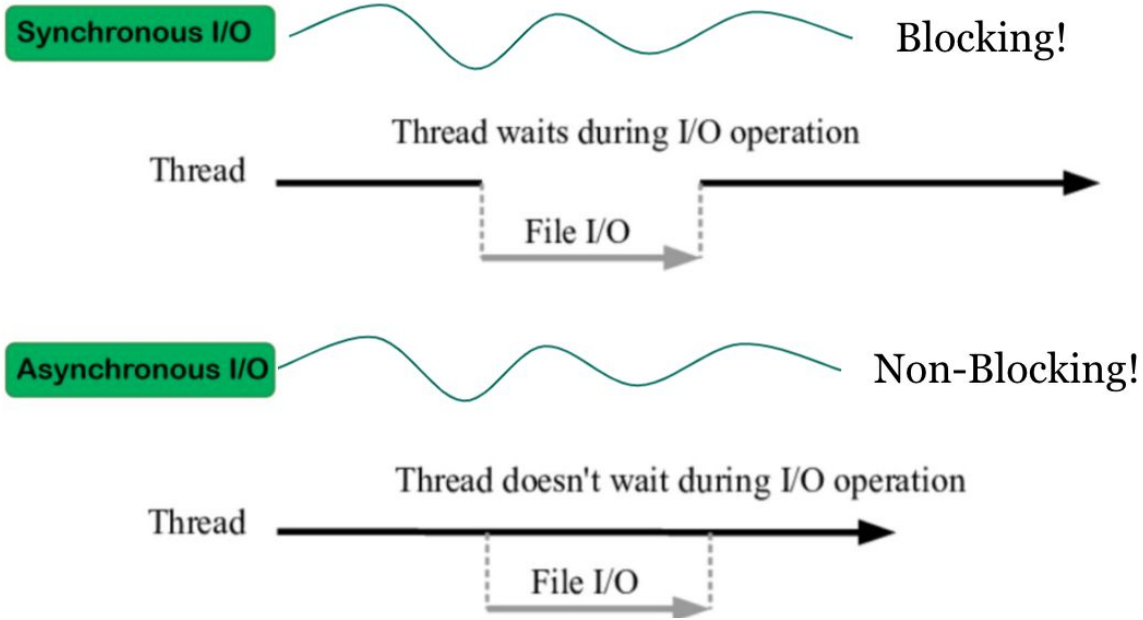
It is the fastest javascript engine.

The V8 engine converts the javascript code into the machine code which the computer actually understands. The result is then generated and returned to node.js.



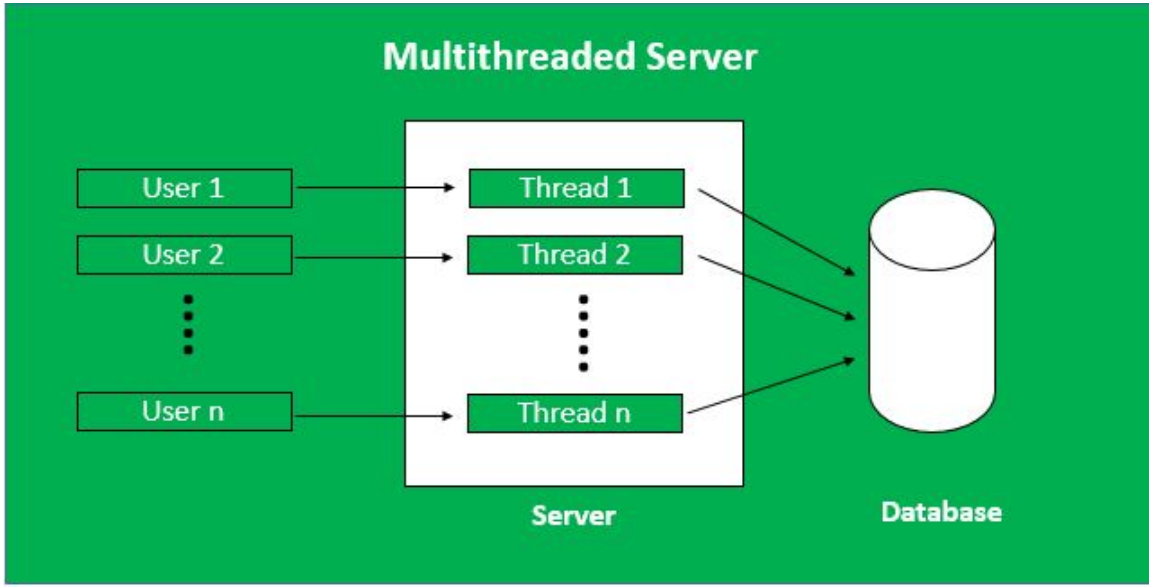
Asynchronous I/O

The Asynchronous I/O allows applications to overlap processing with I/O operations. In simple words, the goal is that the program should never block



Event Loop

Generally, the server-side technologies like PHP, ASP.NET, Ruby & Java Servers all follow a multi-threaded model. In this traditional architectural approach, each client request creates a new thread or a process.



Event Loop

Node js in contrast works differently:

When we use Node.js on a computer, it means that there is a node process running on that computer. The process is just a program in execution.

Now in that process, Node.js runs in a single thread.

A thread is basically just a sequence of instructions.

Therefore, if we have 4 different tasks then all these four tasks will happen in one single thread.

Event Loop

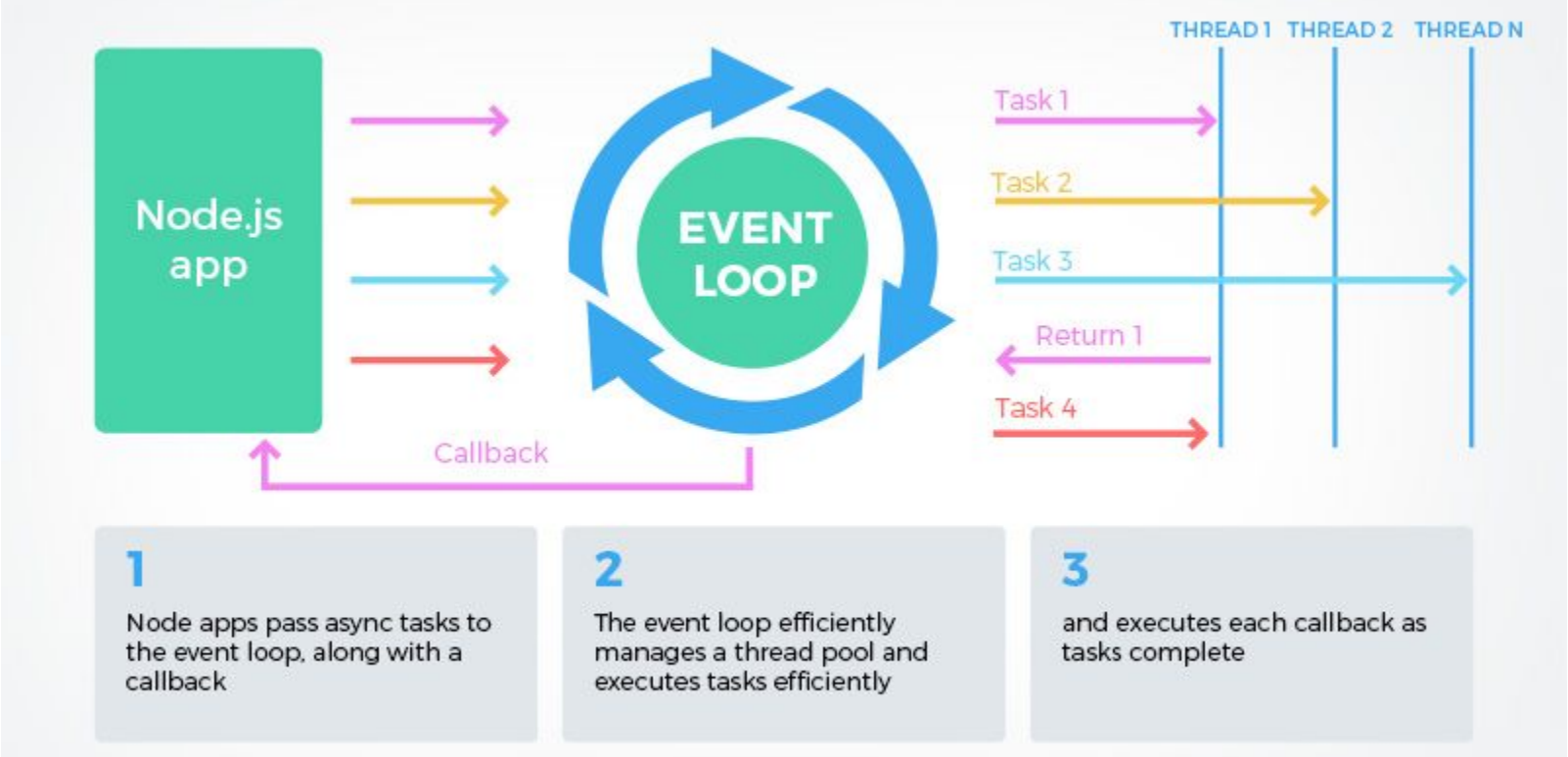
The event loop is called the heart of the node.js.

It executes all the callback functions(functions that are called as soon as some work is finished) in a single thread and it also offloads heavy or expensive tasks like compressing a file to a thread pool.

Eventloop makes asynchronous programming possible in node.js.

So, this single thread doesn't have to wait for the request to complete and is free to handle the next request. When asynchronous I/O work completes then it processes the request further and sends the response.

Event Loop


THREAD 1
THREAD 2
THREAD N
Node.js app
EVENT LOOP
Task 1
Task 2
Task 3
Return 1
Task 4
Callback
1
Node apps pass async tasks to the event loop, along with a callback
2
The event loop efficiently manages a thread pool and executes tasks efficiently
3
and executes each callback as tasks complete

Event-Driven Architecture

In Node, there are certain objects called event emitters that emit named events as soon as something important happens in the app, like a request hitting server or a file finishing to read.

These events are then picked up by event listeners that we developers set up, which will fire off functions(callback functions) that are attached to each listener.

The event-driven architecture makes it way more straight forward to react multiple times to the same event.

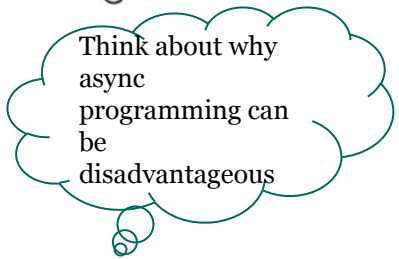
Pros and Cons

Pros

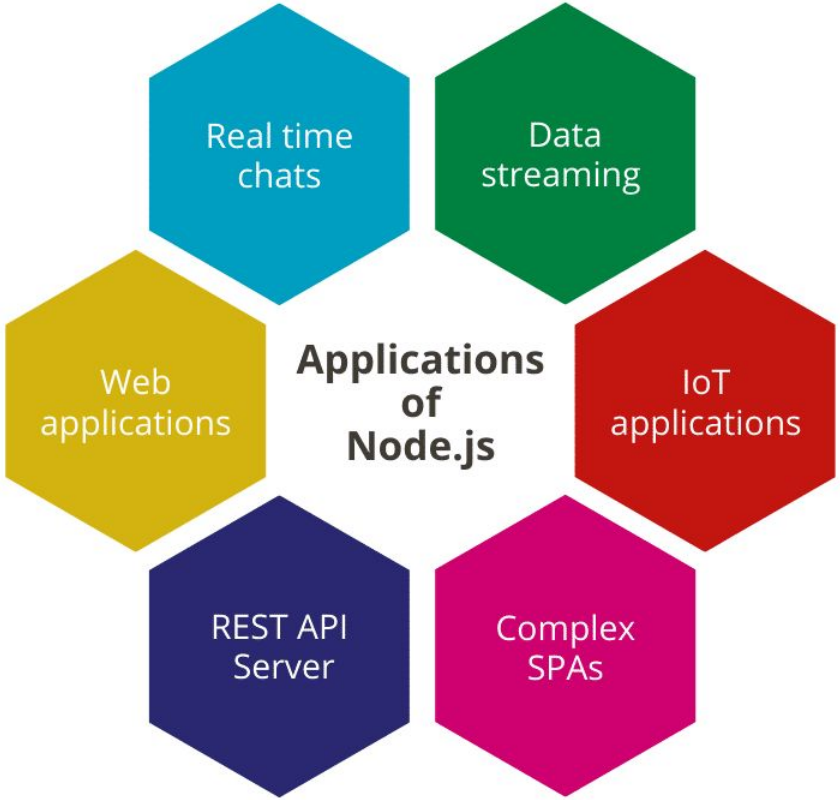
- Used to build scalable web applications
- Works well with MongoDB
- Easy to learn
- Helps to create highly performant applications
- Growing community

Cons

- Based on JavaScript which is dynamically typed making it harder to debug
- APIs used in Node undergo changes frequently
- Uses asynchronous programming

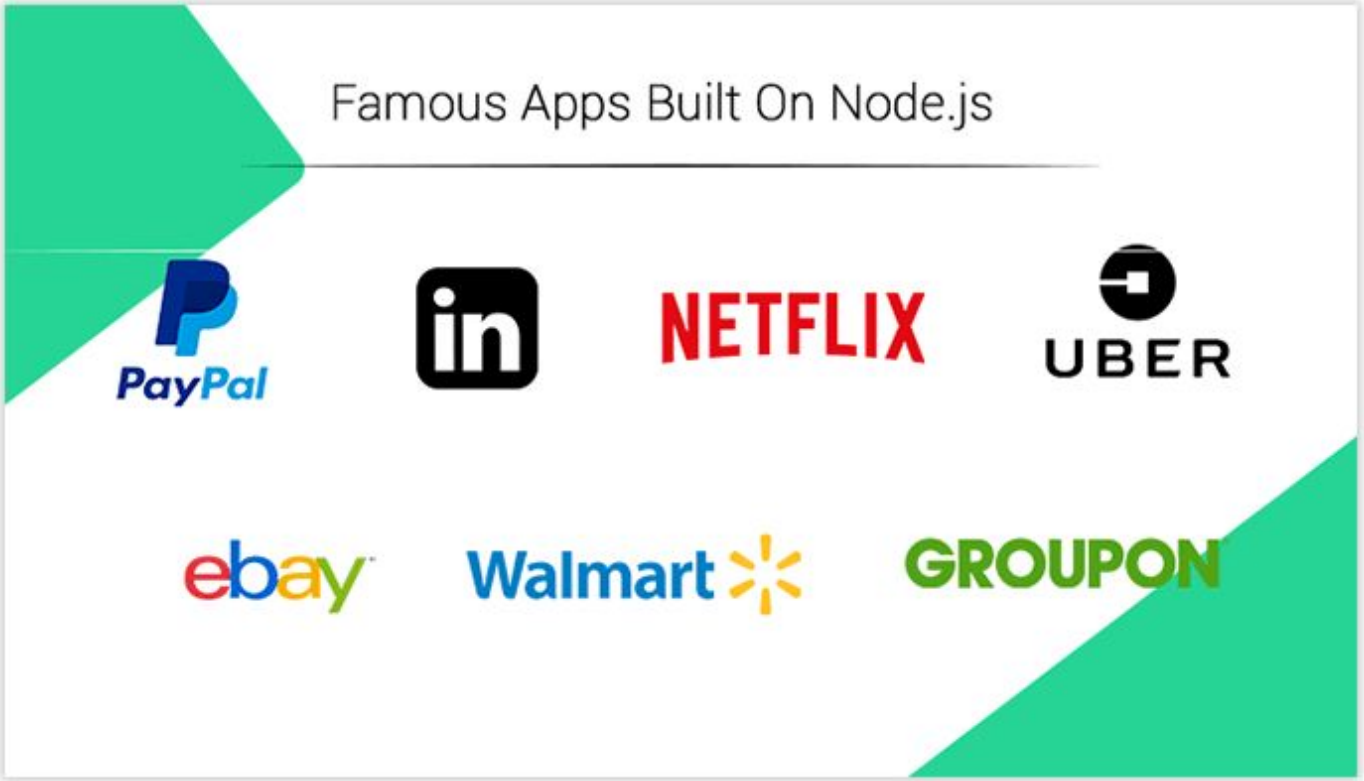


Applications of NodeJS



It is not advisable to use Node.js for CPU intensive applications!

NodeJS in Use



Setup

Installation of NodeJS is straightforward using the installer package available at NodeJS official website.
Have an IDE ready; Visual Studio is highly recommended.

- Download the installer from NodeJS WebSite: **https://nodejs.org/en/**
- Run the installer.
- Follow the installer steps, agree the license agreement and click the next button.
- Restart your system/machine.

Now, test NodeJS by printing its version using the following command in Command Prompt:

```
node -v
```

Test npm by printing its version using command

```
npm -v
```

Have an IDE ready; Visual Studio is highly recommended.

Welcome to NodeJS

Add the following to a file say “welcome.js”:

```
console.log("Welcome to NodeJS!");
```

Run the following line on command prompt:

```
node welcome.js
```

Output:

```
Welcome to NodeJS!
```

Node Modules

Node modules provide a way to re-use code in your Node application.

In Node. js, Modules are **the blocks of encapsulated code** that communicates with an external application on the basis of their related functionality. Modules can be a single file or a collection of multiples files/folders.

These modules have a unique context, thus, they never interfere nor pollute the scope of other modules.

Node.js includes three types of modules:

- Core Module
- Local Module
- Third Party Module

Core Modules

Core Modules are the ones that are provided by node.js itself and loads automatically when Node.js process starts. However, it is required to import these modules using the syntax:

```
var module = require('module_name');
```

Core Module	Description
http	Contains classes, methods, and events required to create Node.js HTTP server
url	Contains methods for URL resolution and parsing in Node
querystring	Contains methods to deal with a query string of Node
path	Contains methods to deal with file paths
fs	Contains classes, methods, and events to work with file I/O
util	Contains utility functions that can be useful for programmers

Some Core modules

```
const path = require('path');
 //Use the core library file.
console.log("Directory of index file: "
+ path.basename(__filename));
console.log("Extension of index file: "
+ path.extname(__filename));
```

Output:

```
Directory of index file: coremodule_1.js
Extension of index file: .js
```

```
var util = require('util');
var txt = 'Congratulate %s on his %dth
birthday!';
var result = util.format(txt, 'Linus',
6);
console.log(result);
```

Output:

```
Congratulate Linus on his 6th birthday!
```

Local Module

The local modules of Node.js are custom modules that are created locally by user/developer in the application.

Local_module.js

```
var detail = {
   name: function (name) {
     console.log('Name: ' + name);
   },
   domain:function (domain) {
     console.log('Domain: ' +
domain);
   }
  };
  module.exports = detail;
```

exported

app.js

```
var myLogModule =
require('./Local_module.js');
myLogModule.name('School');
myLogModule.domain('Education');
```

exports is a special object to expose a module to another application.
So, whatever you assign to module.exports will be exposed as a module.

More on Local Modules

myfirstmodule.js

```
obj={
    firstname: "Jane",
    lastname: "Doe"
};
module.exports.obj=obj;
//module chaining is also possible
module.exports.upper=function(name){
    return name.toUpperCase();
}
module.exports.mydatetime=function(){
    return Date();
};
```

app.js

```
var custom_module =
require("./myfirstmodule.js");
console.log(custom_module.mydatetime());
console.log(custom_module.obj);
console.log(custom_module.upper("jane
doe"));
```

You can export simple literals ,strings or objects!

Output:

```
Sun Oct 31 2021 01:34:30 GMT+0530 (India Standard Time)
{ firstname: 'Jane', lastname: 'Doe' }
JANE DOE
```

Third Party Modules

Third party modules can be downloaded from NPM registry that can help make coding better.

These modules are generally developed by other developers and are free to use.

Some of the best known packages include:

- Express
- Rectify
- Lodash
- Mongoose, etc

Third party modules can be install inside the project folder or globally, using npm.

Third Party Modules: NPM

NPM stands for Node Package Manager which as the name suggests is a package manager for Node.js packages/modules. From Node version 0.6.0. onwards, npm has been added as default in the node installation.

It saves you from the hassle of installing npm explicitly.

Globally Loading the 3rd party module:

```
npm install --g <module_name>
```

Include your module file in your main application file:

```
npm install --save <module_name>
```

File System Module

The file system module, or simply fs, allows you to access and interact with the file system on your machine.

Since it is a core module, all you have to do is import it!

Using the fs module, you can perform actions such as:

- Reading files
- Creating files
- Updating files
- Deleting files
- Renaming files

File System Module: Async vs Sync

It's important to note that by default, all the fs methods are asynchronous. However, you can use the synchronous version by adding Sync at the end of the method.

For instance, a method such as writeFile becomes writeFileSync.

Synchronous methods complete the code synchronously, and thus they block the main thread.

Blocking the main thread in Node.js is considered bad practice.

Asynchronous methods take the last parameter as the completion function callback and the first parameter of the callback function as error.

Read from a File

To read the physical file asynchronously use:

```
fs.readFile(fileName[,options], callback)
```

Where:

- filename: Full path and name of the file as a string.
- options: The options parameter can be an object or string which can include encoding and flag. The default encoding is utf8 and default flag is "r".
- callback: A function with two parameters err and fd. This will get called when readFile operation completes.

```
const fs = require('fs')

fs.readFile('./test.txt', 'utf8' , (err,
data) => {
if (err) {
  console.error(err)
  return
}
console.log(data)
})
```

Shorthand!

Write to a File

To write to a physical file asynchronously use:

```
fs.writeFile(filename,data[,options],
callback)
```

Where:

- filename: Full path and name of the file as a string.
- data: The content to be written in a file.
- options: Object or string which can include encoding, mode and flag. The default encoding is utf8 and default flag is "r".
- callback: A function with two parameters err and fd. This will get called when write operation completes.

```
const fs = require('fs')
const content = 'Some content!'


fs.writeFile('./test.txt', content,
err => {
if (err) {
  console.error(err)
  return
}
//file written successfully
})
```

Opening a File

Alternatively, you can open a file for reading or writing using fs.open() method.

```
fs.open(path, flags[, mode], callback)
```

Where:

- path: Full path with name of the file as a string.
- flag: The flag to perform operation
- mode: The mode for read, write or readwrite. Defaults to 0666 readwrite.
- callback: A function with two parameters err and fd. This will get called when file open operation completes.

Flags

Flag	Description
r	Open file for reading. An exception occurs if the file does not exist.
r+	Open file for reading and writing. An exception occurs if the file does not exist.
rs	Open file for reading in synchronous mode.
rs+	Open file for reading and writing, telling the OS to open it synchronously. See notes for 'rs' about using this with caution.
w	Open file for writing. The file is created (if it does not exist) or truncated (if it exists).
wx	Like 'w' but fails if path exists.
w+	Open file for reading and writing. The file is created (if it does not exist) or truncated (if it exists).
wx+	Like 'w+' but fails if path exists.
a	Open file for appending. The file is created if it does not exist.
ax	Like 'a' but fails if path exists.
a+	Open file for reading and appending. The file is created if it does not exist.
ax+	Like 'a+' but fails if path exists.

Renaming and Updating Files

To asynchronously rename a file at the given old path to a given new path use:

```
fs.rename( oldPath, newPath, callback )
```

It will overwrite the destination file if it already exists.

```
const fs = require('fs');
fs.rename('test.txt', 'world.txt', () =>
{
console.log("File Renamed!");
});
```

To asynchronously append the specified content at the end of the specified file use:

```
fs.appendFile( path, data[, options],
callback )
```

```
var fs = require('fs');

fs.appendFile('mynewfile1.txt', ' This
is my text.', function (err) {
 if (err) throw err;
 console.log('Updated!');
});
```

File System Module: Deleting and Closing Files

To asynchronously delete an existing file use:

```
fs.unlink(path, callback);
```

It will overwrite the destination file if it already exists.

```
var fs = require('fs');
fs.unlink('test.txt', function () {
  console.log('operation complete.');
});
```

To close a file descriptor use:

```
fs.close(fd, callback);
```

Where:

- fd – This is the file descriptor returned by file fs.open() method.
- callback – This is the callback function. No arguments other than a possible exception are given to the completion callback.

```
const fs = require('fs');
fd = fs.openSync("test.txt");
fs.close(fd, (err) => {
if (err)
   console.error('Failed to close file',
err);
else {
   console.log(" File Closed
successfully");}
});
```

Some Important fs Methods

Method	Description
fs.readFile(fileName [,options], callback)	Reads existing file.
fs.writeFile(filename, data[, options], callback)	Writes to the file. If file exists then overwrite the content otherwise creates new file.
fs.open(path, flags[, mode], callback)	Opens file for reading or writing.
fs.rename(oldPath, newPath, callback)	Renames an existing file.
fs.chown(path, uid, gid, callback)	Asynchronous chown.
fs.stat(path, callback)	Returns fs.stat object which includes important file statistics.
fs.link(srcpath, dstpath, callback)	Links file asynchronously.
fs.symlink(destination, path[, type], callback)	Symlink asynchronously.
fs.rmdir(path, callback)	Renames an existing directory.
fs.mkdir(path[, mode], callback)	Creates a new directory.
fs.readdir(path, callback)	Reads the content of the specified directory.
fs.utimes(path, atime, mtime, callback)	Changes the timestamp of the file.
fs.exists(path, callback)	Determines whether the specified file exists or not.
fs.access(path[, mode], callback)	Tests a user's permissions for the specified file.
fs.appendFile(file, data[, options], callback)	Appends new content to the existing file.

Buffer Module

In Node.js, buffers are a special type of object that can store raw binary data. A buffer represents a chunk of memory - typically RAM - allocated in your computer. Once set, the size of a buffer cannot be changed.

A buffer stores bytes. A byte is a sequence of eight bits. Bits are the most basic unit of storage on your computer, they can hold the value of either 0 or 1.

Node.js exposes the Buffer class in the global scope (you don't need to import or require it like other modules). With this API, you get a series of functions and abstractions to manipulate raw binaries.

Node.js displays bytes using the hexadecimal system!!

```
<Buffer 61 2e 71 3b 65 2e 31 2f 61 2e>
```

Creating a Buffer: Buffer.alloc()

The .alloc() method is useful when you want to create empty buffers, without necessarily filling them with data. By default, it accepts a number and returns a buffer of that given size filled with zeroes:

```
Buffer.alloc(size, [fill, encoding]);
```

```
Buffer.alloc(6);
  // --> <Buffer 00 00 00 00 00 00>
```

You can later on fill the buffer with any data you want:

```
Buffer.alloc(6, "x", "utf-8");
  // --> <Buffer 78 78 78 78 78 78>
```

You can also fill the buffer with other content than 0 and a given encoding:

```
// Creates a buffer of size 1 filled
with 0s (<Buffer 00>)
const buff = Buffer.alloc(1);

// Fill the first (and only) position
with content
buff[0] = 0x78 // 0x78 is the letter "x"

console.log(buff.toString('utf-8');
  // --> 'x'
```

Creating a Buffer: Buffer.allocUnsafe()

With .allocUnsafe() , the process of sanitizing and filling the buffer with zeroes is skipped. The buffer will be allocated in a area of memory that may contain old data (that's where the "unsafe" part comes from).

```
Buffer.allocUnsafe(size);
```

```
// Allocates a random area of memory with size 10000
// Does not sanitizes it (fill with 0) so it may contain
old data
const buff = Buffer.allocUnsafe(10000);

// Prints loads of random data
  console.log(buff.toString("utf-8"));
```

Reading a Buffer

The toString() method returns the buffer object according to the specified encoding.

```
buffer.toString([encoding, start, end]);
```

```
var buffer = new Buffer.alloc(5);
for (var i = 0; i < 5; i++) {
   buffer[i] = i + 97;
}
console.log(buffer.toString());
console.log(buffer.toString('utf-8', 1, 4));
console.log(buffer.toString('hex'));
```

Writing to a Buffer

By default, buffer.write() will write a string encoded in utf-8 with no offset (starts writing from the first position of the buffer). It returns a number, which is the number of bytes that were written in the buffer:

```
buffer.write(value, start, bytes, encoding);
```

```
const buff = Buffer.alloc(9);

buff.write("hey there"); // returns 9 (number of bytes
written)

// If you write more bytes than the buffer supports,
// your data will truncated to fit the buffer.
buff.write("hey christopher"); // returns 9 (number of bytes
written)

console.log(buff.toString());
  // -> 'hey chris'
```

Some Buffer Methods

Method	Description
byteLength()	Returns the numbers of bytes in a specified object
compare()	Compares two Buffer objects
concat()	Concatenates an array of Buffer objects into one Buffer object
copy()	Copies the specified number of bytes of a Buffer object
entries()	Returns an iterator of "index" "byte" pairs of a Buffer object
equals()	Compares two Buffer objects, and returns true if it is a match, otherwise false
fill()	Fills a Buffer object with the specified values
from()	Creates a Buffer object from an object (string/array/buffer)
includes()	Checks if the Buffer object contains the specified value. Returns true if there is a match, otherwise false

Streams

Streams are a way to handle reading/writing files, network communications, or any kind of end-to-end information exchange in an efficient way.

Instead of a program reading a file into memory all at once like in the traditional way, streams read chunks of data piece by piece, processing its content without keeping it all in memory.

Let's take a "streaming" services such as YouTube or Netflix for example: these services don't make you download the video and audio feed all at once. Instead, your browser receives the video as a continuous flow of chunks, allowing the recipients to start watching and/or listening almost immediately

Why Streams?

Streams basically provide two major advantages compared to other data handling methods:

- **Memory efficiency:** you don't need to load large amounts of data in memory before you are able to process it
- **Time efficiency:** it takes significantly less time to start processing data as soon as you have it, rather than having to wait with processing until the entire payload has been transmitted

Streams in NodeJS

- **Writable:** streams to which we can write data. For example, fs.createWriteStream() lets us write data to a file using streams.
- **Readable:** streams from which data can be read. For example: fs.createReadStream() lets us read the contents of a file.
- **Duplex:** streams that are both Readable and Writable. For example, net.Socket
- **Transform:** streams that can modify or transform the data as it is written and read. For example, in the instance of file-compression, you can write compressed data and read de-compressed data to and from a file. Node.js comes with a variety of transform streams in the Core API:
    - zlib - for gzip compressing and decompressing
    - crypto - for encrypting, decrypting, and calculating message digests

Streams in NodeJS

Readable Streams	Writable Streams
HTTP responses, on the client	HTTP requests, on the client
HTTP requests, on the server	HTTP responses, on the server
fs read streams	fs write streams
zlib streams	zlib streams
crypto streams	crypto streams
TCP sockets	TCP sockets
child process stdout and stderr	child process stdin
process.stdin	process.stdout, process.stderr

Events in Streams

The streams throw several events since they are eventEmitter instances. These events are used to track and monitor the stream.

Some of the most commonly used events are:

- Data - Data event is emitted when readable data is available.
- Finish - Finish event is emitted when the stream is done writing data.
- Error - Error event is emitted when an error occurs while reading/writing data.
- End - End event is emitted when the read stream has finished reading data.

Reading from a Stream

```
const fileSystem = require("fs");
var data = "";

const readStream = fileSystem.createReadStream("input.txt");
readStream.on("data", (chunk) => {
   data += chunk;
});

readStream.on("end", () => {
   console.log(data);
});

readStream.on("error", (error) => {
   console.log(error.stack);
});
```

Writing to a Stream

```
const fileSystem = require("fs");
var data = "Sample text";

const writeStream = fileSystem.createWriteStream("output.txt");

writeStream.write(data, "UTF8");

writeStream.end()

writeStream.on("finish", () => {
   console.log("Finished writing");
});

writeStream.on("error", (error) => {
   console.log(error.stack);
});
```

Piping Streams

Piping is a mechanism that involves using the output of another stream input of the other.

```
const fileSystem = require("fs");

const readStream = fileSystem.createReadStream("input.txt");
const writeStream = fileSystem.createWriteStream("output.txt");

readStream.pipe(writeStream);

console.log("Program finished");
```

Transforming a Stream

Compression:

```
var fs=require("fs");

var zlib= require('zlib');
fs.createReadStream('test.txt')  //reads
.pipe(zlib.createGzip())  //compresses
.pipe(fs.createWriteStream("data.txt.gz"));
//writes into the file
console.log("File is compressed"); //zip
file is created
```

Decompression:

```
var zlib= require('zlib');
var fs=require("fs");

fs.createReadStream('data.txt.gz') //reads
.pipe(zlib.createGunzip())  //decompresses
.pipe(fs.createWriteStream("data.txt"));
//writes
console.log("File is decompressed");
```

Callbacks

Callback is an asynchronous equivalent for a function. A callback function is called at the completion of a given task.

Node makes heavy use of callbacks. All the APIs of Node are written in such a way that they support callbacks.

For example, a function to read a file may start reading file and return the control to the execution environment immediately so that the next instruction can be executed. Once file I/O is complete, it will call the callback function while passing the callback function, the content of the file as a parameter. So there is no blocking or wait for File I/O. This makes Node.js highly scalable, as it can process a high number of requests without waiting for any function to return results.

Blocking I/O

```
var fs = require("fs");
var data =
fs.readFileSync('data.txt');

console.log(data.toString());
console.log("Reading Complete");
```

Output:

```
Some content!
Reading Complete
```

The program blocks until it reads the file and then only it proceeds to end the program.

A blocking program executes very much in sequence. From the programming point of view, it is easier to implement the logic for a blocking program.

Non-Blocking I/O

```
var fs = require("fs");

fs.readFile('data.txt', function
(err, data) {
  if (err) return console.error(err);
  console.log(data.toString());
});

console.log("Reading Complete");
```

Output:

```
Reading Complete
Some content!
```

The program does not wait for file reading and proceeds to print "Reading complete" and at the same time, the program continues reading the file.

Non-blocking programs do not execute in sequence

HTTP Module

Node.js has a built-in (core) module called HTTP, which allows Node.js to transfer data over the Hyper Text Transfer Protocol (HTTP).

The HTTP module creates a HTTP server that listens to server ports. The server created can read HTTP requests made by a client through a browser or console.

Let's look at how to build a simple server using the http module.

Step 1:
Create a new file called server.js and include the http module by using the require() function

```
const http = require('http');
```

There is also a HTTPS module for secure HTTP requests

HTTP Module

```
const http=require('http');
const server = http.createServer((req,
res) => {
   if (req.url === '/') {
       res.write('<h1>HTTP Server says
hi!</h1>');
   }
   res.end();
});
server.listen(5000);
console.log('The HTTP Server is running on
port 5000');
```

Step 2:
Create an HTTP server using the createServer() method of the http object.
The createServer() accepts a callback that has two parameters:
HTTP request (req) and response (res).
Inside the callback, we send an HTML string to the browser if the URL is '/' and end the request.

Step 3:
Listen to the incoming HTTP request on the port 5000

Step 4:
Verify that the server is up and running on localhost:3000

Another Simple Server

```
var http = require('http'); // Import Node.js core module
var server = http.createServer(function (req, res) {   //create web server
   if (req.url == '/') { //check the URL of the current request

       // set response header
       res.writeHead(200, { 'Content-Type': 'text/html' });

       // set response content
       res.write('<html><body><p>This is home Page.</p></body></html>');
       res.end();
   }
   else if (req.url == "/student") {

       res.writeHead(200, { 'Content-Type': 'text/html' });
       res.write('<html><body><p>This is student Page.</p></body></html>');
       res.end();
  //to be continued
```

Header as an object

status code

res.write sends a chunk of the response body

Another Simple Server

```
    //continued
    else if (req.url == "/admin") {
        res.writeHead(200, { 'Content-Type': 'text/html' });
        res.write('<html><body><p>This is admin Page.</p></body></html>');
        res.end();

    }
    else
        res.end('Invalid Request!');

});
server.listen(8081);

console.log('Node.js web server at port 8081 is running..');
```

signals to the server that all of the response headers and body have been sent

Server listening at http://localhost:8081/
Observe what happens when you give http://localhost:8081/student and http://localhost:8081/admin !!

Handling HTTP Requests

So far we have made requests to the server through a browser by just changing the URL. This is however not very practical.

Usually an application initiates a HTTP request and the server is informed about the request.

The server then decodes the HTTP request and sends it to the corresponding application or server for further processing.

For the server we created, let's look at how to send a request for '/admin' from a client application.

Handling HTTP Requests

The common programming task of making a HTTP request to a web server can be easily performed using the default http module with the help of http.request

```
http.request(options, callback)
```

This method is used to issue a HTTP request where:

- Options is an object that specifies host, port, path and other header information.
  - host: the domain or IP address of the server
  - port: the port (e.g. 80 for HTTP)
  - path: the request path, including the query string (e.g. 'index.html?page=12')
- The callback passed to the method will receive a http.ClientResponse object when the request is made. The ClientResponse is a Readable Stream.

Handling HTTP Requests: Client

```
const http = require('http');
const options = {
   hostname: 'localhost',
   port: 8081,
   path: '/admin',
   method: 'GET'
};
var callback= function(response){
   var body="";
   response.on('data', function(data){
       body+=data;
   });
response.on('end', function(){
   console.log(body);
});
response.on('error', (err) => {
   console.error(err);});
};

const req = http.request(options, callback);
req.end();
```

the resource under request . In this case, it could be / or / admin or /student

To make a POST requests , just change method to 'POST"

Binding events to their event handlers. Recall node events...

Note: Run your server first. Open up a new terminal and run your client

Working with Queries: Server

Real world applications will also include search strings. Let us try to handle them!

```
var http = require('http');
var url = require('url');

http.createServer(function (req, res) {
 res.writeHead(200, {'Content-Type': 'text/html'});
 var q = url.parse(req.url, true).query;

 var txt = q.name + " " + q.srn;
 res.write(txt);
 res.end();
}).listen(8080);
```

The core URL module splits up a web address into readable parts.

Query parameters

url.parse() will create an object to easily access url parts as properties.

Eg:
url.parse(req.url).host ⟶ returns hostname
url.parse(req.url).query ⟶ returns object containing query parameters like so {q: 'val'}

Working with Queries: Client

```
const http = require('http');
const options = {
   hostname: 'localhost',
   port: 8080,
   path: '/?name=ABC&srn=PES000',
   method: 'GET'
};
var callback= function(response){
   var body="";
   response.on('data', function(data){
       body+=data;
   });
response.on('end', function(){
   console.log(body);
});
response.on('error', (err) => {
   console.error(err);
});};
const req = http.request(options, callback);
req.end();
```

Query string wherein the parameter=value pairs are separated by '&'. These parameters are used on the server side

The Data Store

If your application stores any data (user profiles, content, comments, uploads, events, etc.), then you're going to want to use a data store.

Not only do they allow you to store, search, filter and present information based on web requests from users, they also allow a wide variety of mathematical and statistical calculations on queries submitted from web browsers.

So, in MERN stack, this database tier is developed using :

MongoDB

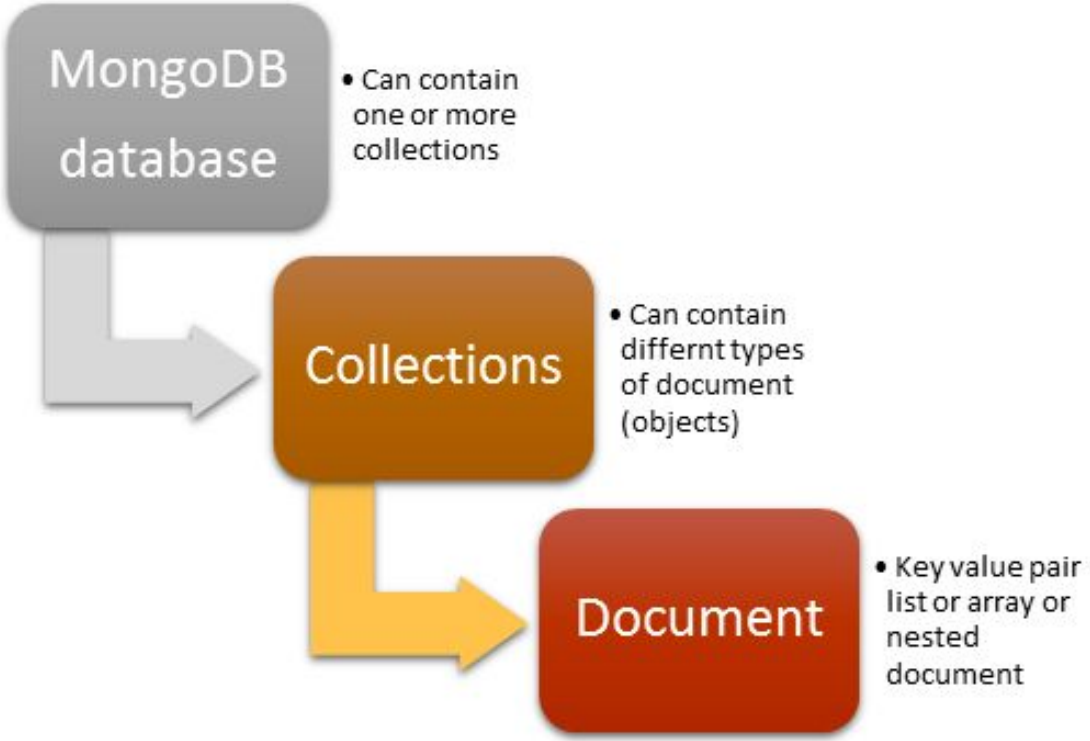
MongoDB is the database used in the MERN stack.

It is a NoSQL (non-relational) document-oriented database, meaning it's essentially **not** a conventional database where you have tables with columns and rows and strict relationships among them

It makes use of a flexible schema and a JSON-based query language.

Not only do many modern companies (including Facebook and Google) use MongoDB in production, but some older established companies such as SAP and Royal Bank of Scotland have adopted MongoDB

Components of MongoDB Architecture



MongoDB: Documents

MongoDB documents are composed of field-and-value pairs and have the following structure:

```
{
   field1: value1,
   field2: value2,
   ...
   fieldN: valueN
}
```

Example

```
{
  name: "sue",                          <-- field: value
  age: 26,                              <-- field: value
  status: "A",                          <-- field: value
  groups: [ "news", "sports" ]          <-- field: value
}
```

MongoDB: Collections

MongoDB stores documents in collections.
Collections are analogous to tables in relational databases.
Collections do not enforce a schema, and documents in the same collection can have different fields.

MongoDB: Collections

A primary key is mandated in MongoDB, and it has the reserved field name _id.

Even if _id field is not supplied when creating a document, MongoDB creates this field and auto-generates a unique key for every document.

MongoDB: Database

In MongoDB, databases hold one or more collections of documents

It is a logical grouping of many collections.

A database connection is restricted to accessing only one database, so to access multiple databases, multiple connections are required.

Thus, it is useful to keep all the collections of an application in one database, though a database server can host multiple databases.

Drawing Parallels

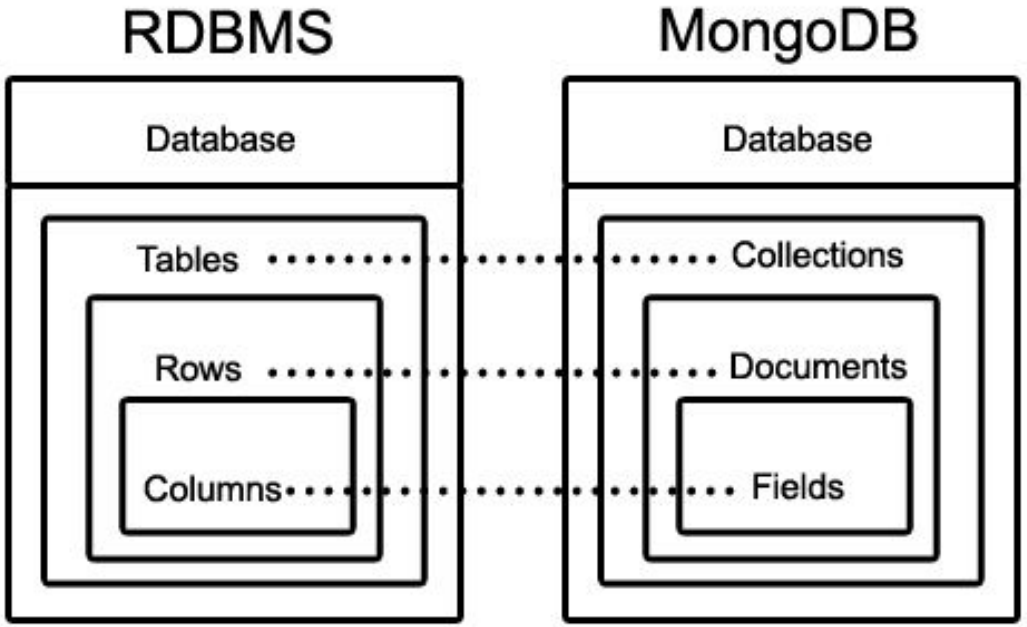


Table vs Collection

Relational Database

Student_Id	Student_Name	Age	College
1001	Chaitanya	30	Beginnersbook
1002	Steve	29	Beginnersbook
1003	Negan	28	Beginnersbook

MongoDB

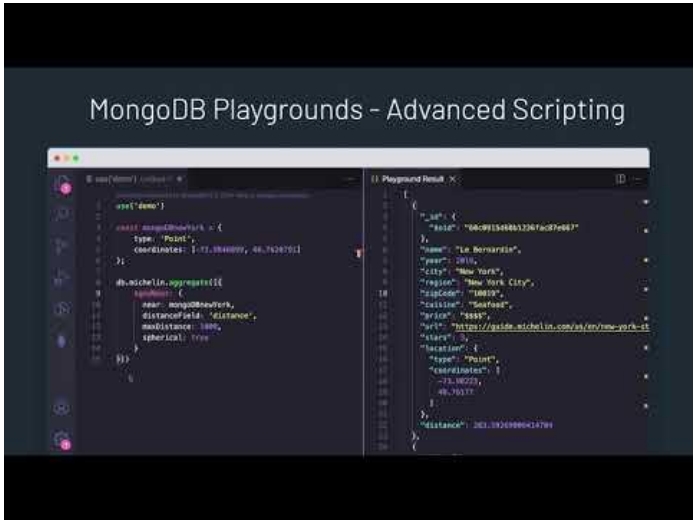
```
{
 "_id": ObjectId("....."),
 "Student_Id": 1001,
 "Student_Name": "Chaitanya",
 "Age": 30,
 "College": "Beginnersbook"
}
{
 "_id": ObjectId("....."),
 "Student_Id": 1002,
 "Student_Name": "Steve",
 "Age": 29,
 "College": "Beginnersbook"
}
{
 "_id": ObjectId("....."),
 "Student_Id": 1003,
 "Student_Name": "Negan",
 "Age": 28,
 "College": "Beginnersbook"
}
```

Documents in the same collection can have different fields but all documents in a collection must have a unique _id!

MongoDB Installation

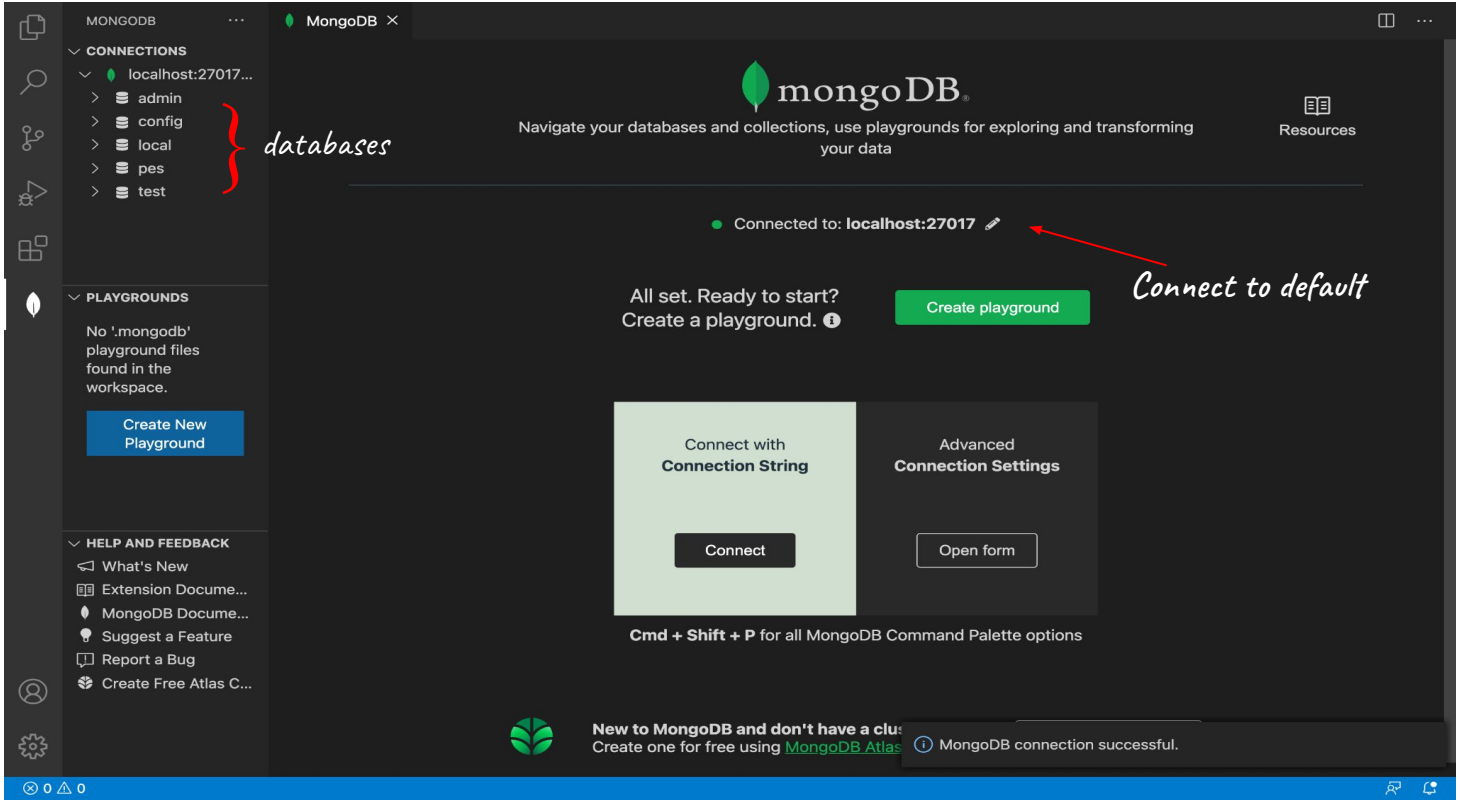
Install mongodb by following the instructions for your respective OS:
https://docs.mongodb.com/manual/installation/

If you are using VS Code, install the MongoDB for VS Code extension. This makes working with MongoDB so much easier!

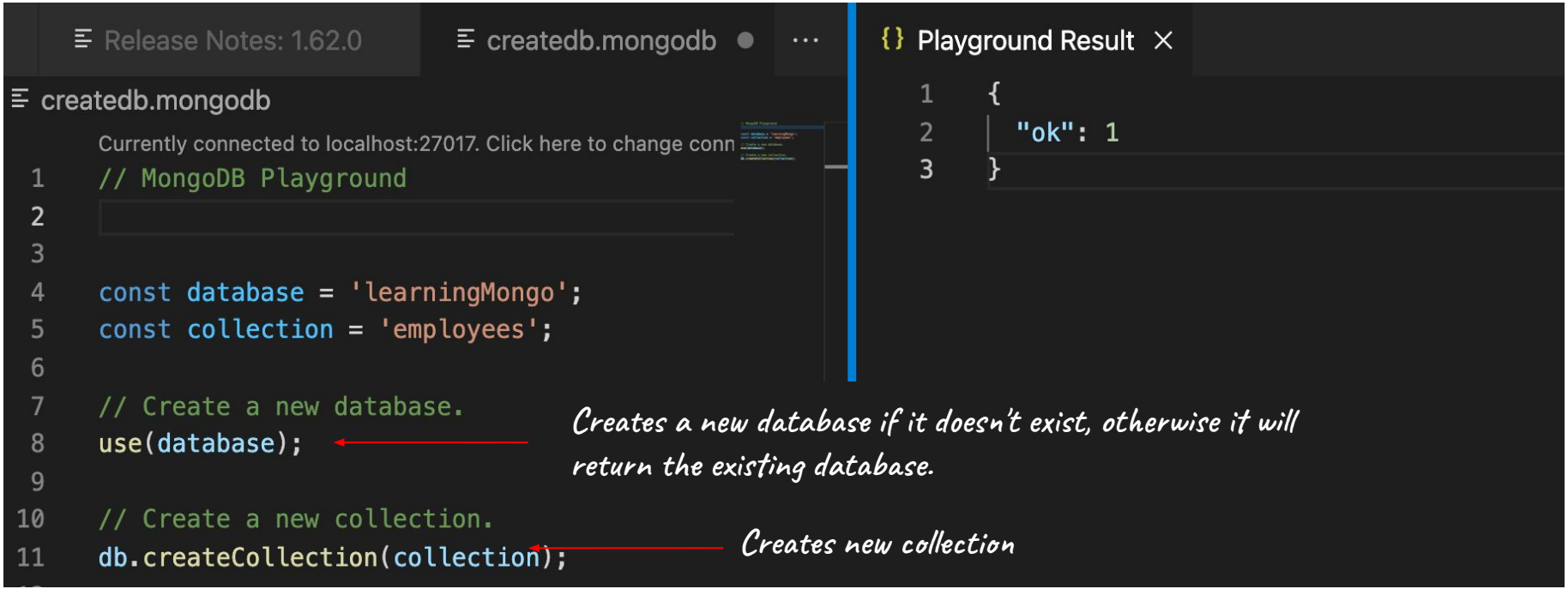


The MongoDB syntax however remains the same even if you wish to work on your console.

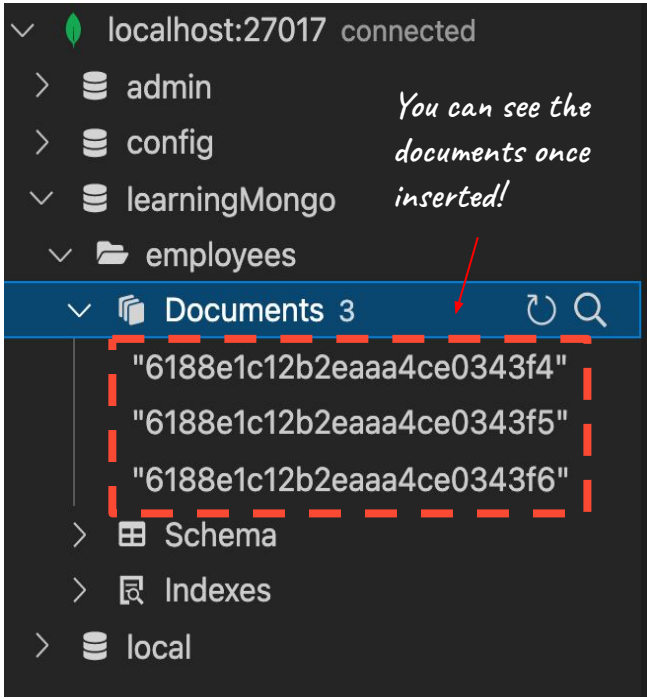
MongoDB for VS Code



Creating Databases and Collections



Creating Documents




```
Currently connected to localhost:27017. Click here to change connection.
1  use("learningMongo");
2  //You must mention which database
3  //you want to perform actions on
4  //inserting one document
5  db.employees.insertOne({
6   id: 1,
7   name: { first: 'John', last: 'Doe' },
8   age: 48
9  })
10 //insert many documents
11 db.employees.insertMany([
12  { id: 3, name: { first: 'Alice', last: 'A' }, age: 32 },
13  { id: 4, name: { first: 'Bob', last: 'B' }, age: 64 },
14 ])
```

Reading from MongoDB

```
Currently connected to localhost:27017. Click here to change cor
/*
The MongoDB find query is an in-built
function which is used to retrieve the
documents in the collection.
*/

use("learningMongo");
db.employees.find();
```

Result:


```
[
  {
    "_id": {
      "$oid": "6188e37b82faeb29809bb537"
    },
    "id": 1,
    "name": {
      "first": "John",
      "last": "Doe"
    },
    "age": 48
  },
  {
    "_id": {
      "$oid": "6188e37b82faeb29809bb538"
    },
    "id": 3,
    "name": {
      "first": "Alice",
      "last": "A"
    },
    "age": 32
  },
  {
    "_id": {
      "$oid": "6188e37b82faeb29809bb539"
    },
    "id": 4,
    "name": {
      "first": "Bob",
      "last": "B"
    },
    "age": 64
  }
]
```


You can see that mongoDB assigns _id for all documents ALWAYS!

Reading from MongoDB

Currently connected to localhost:27017. Click here to change connection.

```
/*
The MongoDB find query is an in-built
function which is used to retrieve the
documents in the collection.
*/

use("learningMongo");
db.employees.find({ id :1 });

/*
filter is an object where the property name
is the field to filter on, and the value is
its value that it needs to match
*/
```

filter

Result:

```
[
  {
    "_id": {
      "$oid": "6188e37b82faeb29809bb537
    },
    "id": 1,
    "name": {
      "first": "John",
      "last": "Doe"
    },
    "age": 48
  }
]
```

Reading from MongoDB

Result:

```
Currently connected to localhost:27017. Click here to change con
/*
The MongoDB find query is an in-built
function which is used to retrieve the
documents in the collection.
*/

use("learningMongo");
db.employees.find({ 'name.last': 'Doe' })
```

```
[
  {
    "_id": {
      "$oid": "6188e37b82faeb29809bb537"
    },
    "id": 1,
    "name": {
      "first": "John",
      "last": "Doe"
    },
    "age": 48
  }
]
```

Reading from MongoDB

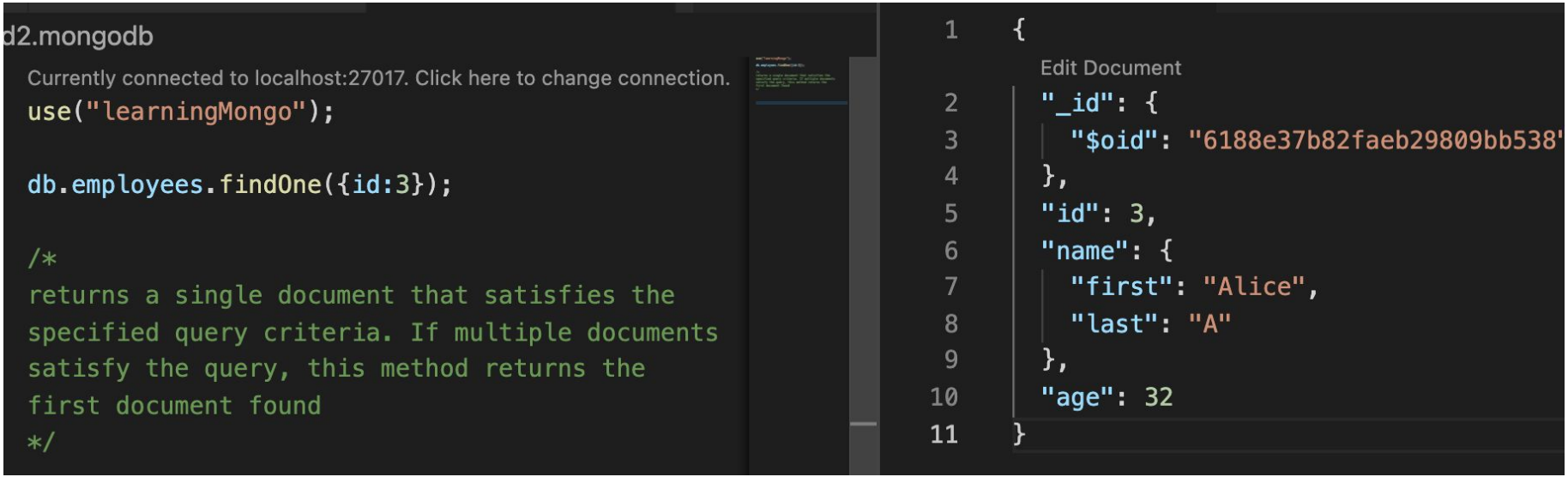
```
Currently connected to localhost:27017. Click here to change connecti
/*
The MongoDB find query is an in-built
function which is used to retrieve the
documents in the collection.
*/

use("learningMongo");
 db.employees.find({ age: { $gte: 50 } });

/*
filters can also look like:
fieldname: { operator: value }.
 $eq --> equal to
 $gt --> greater
 $gte --> greater than or equal to
*/
```

Result:

```
[
  {
    "_id": {
      "$oid": "6188e37b82faeb29809bb539"
    },
    "id": 4,
    "name": {
      "first": "Bob",
      "last": "B"
    },
    "age": 64
  }
]
```

Reading from MongoDB

Result:



Reading from MongoDB

```
1   use('learningMongo');
2   let result=db.employees.find().toArray()
3   /*Now, the variable result should be an array with elements,
4   each an employee document. Using the JavaScript array method forEach()
5   we can iterate through them and print the first names of each employee:
6    */
7   result.forEach((e) => print('First Name:', e.name.first))
8
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
First Name:
John
First Name:
John
First Name:
Jane
```

Updating in MongoDB

d2.mongodb

Currently connected to localhost:27017. Click here to change connection.

```
use("learningMongo");

db.employees.updateOne(
    { id: 1 }, { $set: {age: 23 } }
    );
db.employees.find({id:1});
/*
first argument is a query filter, the same as
the filter that find() takes. The second
argument is an update specification if only
some fields of the object need to be changed
updateOne() stops after finding and updating
the first matching document
*/
```

Finds all documents with id 1 and updates age to 23

Result:

Edit Document

```
[
  {
    "_id": {
      "$oid": "6188e37b82faeb29809bb537"
    },
    "id": 1,
    "name": {
      "first": "John",
      "last": "Doe"
    },
    "age": 23
  }
]
```

Updating in MongoDB

d2.mongodb

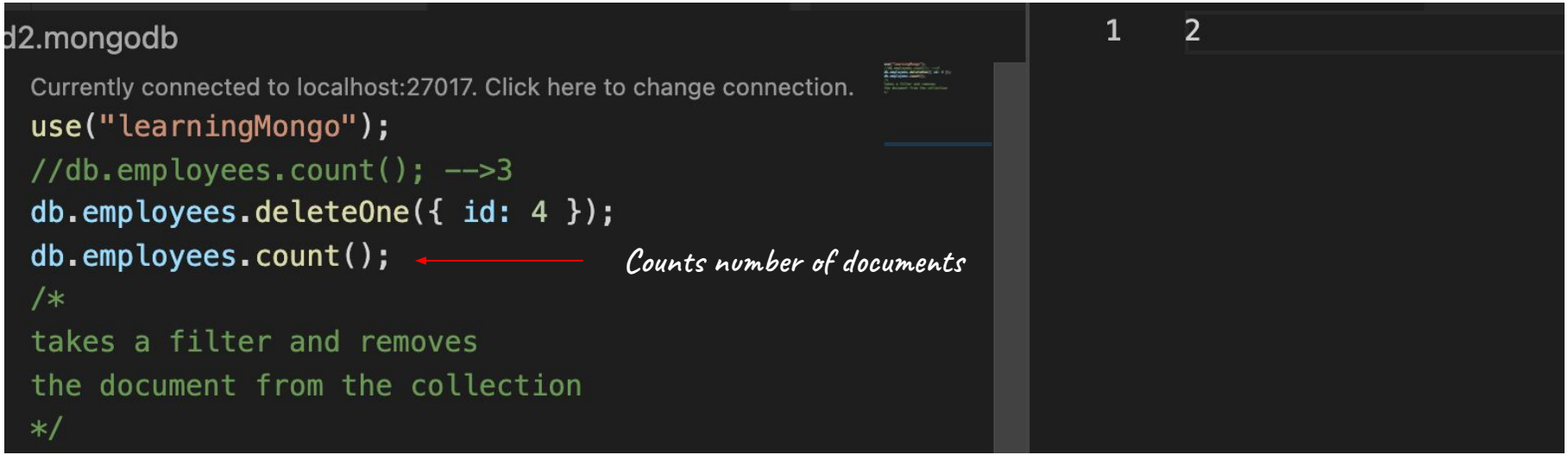
```
Currently connected to localhost:27017. Click here to change connection.
use("learningMongo");
db.employees.updateMany(
    {},
    { $set: { organization: 'MyCompany' } })
db.employees.find();
/*
The format is the same as the updateOne() method,
but the effect is that all documents that match
will be modified.
*/
```

Finds all documents and adds a new field

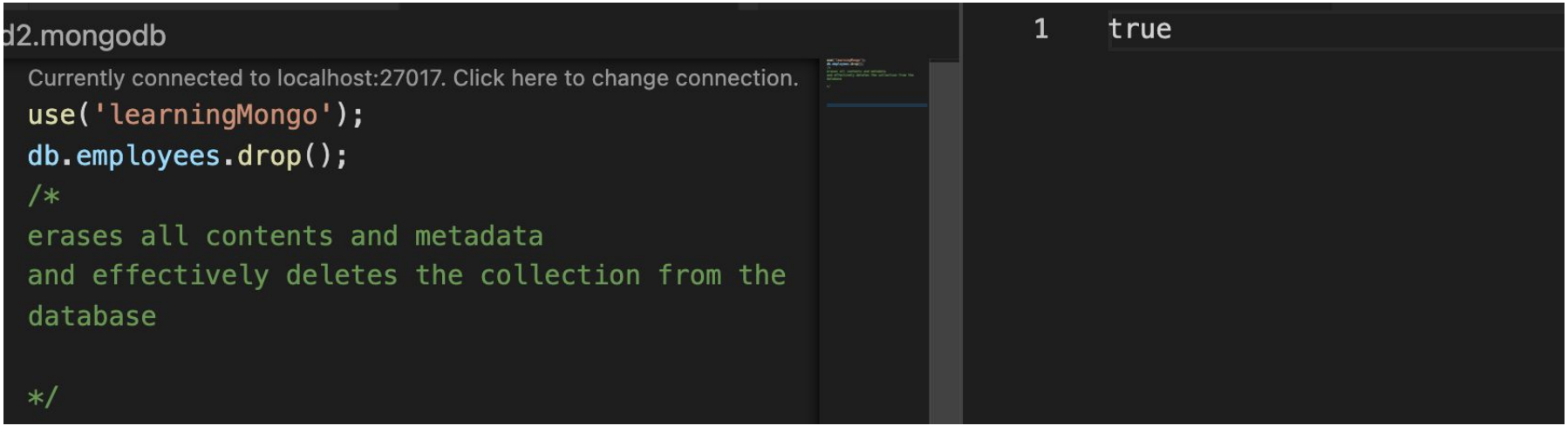
Result:


```
[
  {
    "_id": {
      "$oid": "6188e37b82faeb29809bb537"
    },
    "id": 1,
    "name": {
      "first": "John",
      "last": "Doe"
    },
    "age": 23,
    "organization": "MyCompany"
  },
  {
    "_id": {
      "$oid": "6188e37b82faeb29809bb538"
    },
    "id": 3,
    "name": {
      "first": "Alice",
      "last": "A"
    },
    "age": 32,
    "organization": "MyCompany"
  },
  {
    "_id": {
      "$oid": "6188e37b82faeb29809bb539"
    },
    "id": 4,
    "name": {
      "first": "Bob",
      "last": "B"
    },
    "age": 64,
    "organization": "MyCompany"
  }
]
```

Deleting in MongoDB



Dropping Collection



MongoDB Node.js Driver

In order to access your MongoDB databases and interact with the MongoDB server through NodeJS, you will need a driver: mongodb.

This driver should be installed using npm:

```
npm install mongodb
```

It is an excellent easy-to-use and asynchronous node interface to MongoDB.

To connect to MongoDB, first we import MongoClient from the driver, then we create a new client object from it using a URL that identifies a database to connect to, and finally call the connect method on it.

Let's learn how to use this driver to connect to collection 'employees' in 'learningMongo' database to read and write into it.

Using MongoDB driver : Writing Using Callbacks

```
const MongoClient = require('mongodb').MongoClient;
const url = 'mongodb://localhost/27017';
const client = new MongoClient(url, { useNewUrlParser: true });
client.connect(function(err, client) {
   if (err) {
       console.log(err);
       return;
   }
   const db = client.db("learningMongo");

   const collection = db.collection('employees');

   const employee = { id: 1, name: 'A. Callback', age: 23 };
   collection.insertOne(employee, function(err) {
       if (err) {
           client.close();
           return;
       }
       console.log("Success!!") ;
       });
   client.close();});
```

Default connection settings. If you did not use default connection when creating your db, change the url appropriately.

If successfully connected , the client parameter will contain the initialised db object

Which database to connect to

Which collection to connect to

Document to insert

Inserts one document

Closes connection

Using MongoDB driver : Reading Using Callbacks

```
const MongoClient = require('mongodb').MongoClient;
const url = 'mongodb://localhost/27017';
const client = new MongoClient(url, { useNewUrlParser: true });
client.connect(function(err, client) {
   if (err) {
       console.log(err);
       return;}
   const db = client.db("learningMongo");
   const collection = db.collection('employees');

   collection.find().toArray(function(err, result) {
       if (err) {
           console.log(err)
           client.close();
           return;}
       console.log('Result of find:\n', result);
       client.close();});
});
```

Callback executes after the operation is complete and is required due to the asycn nature of NodeJS!

Returns an array of all the documents read using find()

Using MongoDB driver : async/await paradigm

```
const MongoClient = require('mongodb').MongoClient;
const url = 'mongodb://localhost/27017';
async function testWithAsync(){
   const client = new MongoClient(url, { useNewUrlParser: true });
   try {
       await client.connect();
       console.log('Connected to MongoDB');
       const db = client.db("learningMongo");
       const collection = db.collection('employees');
       await collection.insertOne({ id: 2, name: 'B. Async', age: 16 });
       console.log("Success!");
       const docs = await collection.find().toArray();
       console.log('Result of find:\n', docs);}
   catch(err) {
       console.log(err);}
   finally {
       client.close();
   }
}
testWithAsync();
```

waits until connection is established

await blocks further execution until the current statement has completed executing thus eliminating the need for callbacks!

Running React On Node

Using our knowledge of the Http and fs modules, we can build a simple server to serve a react file just as we would any other file. So keep one of your earlier react codes ready.

```
const http = require('http')
const fs = require('fs')

const server = http.createServer((req, res) => {
 res.writeHead(200, { 'content-type': 'text/html' })
 fs.createReadStream('colours.html').pipe(res)
})

server.listen(8080);
console.log("Server is listening at 8080.......")
```

The react file you want to display

Verify at localhost:8080 or go on to build a client using the examples we've done so far!

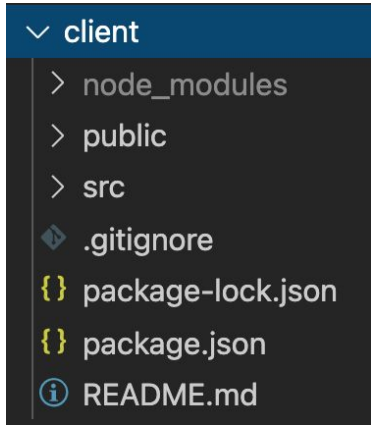
Running React On Node: Creating Your React App

However, the traditional approach to running a React app on NodeJS is with the help of Express. Let's have a look at this approach.

Step 1: Creating the react app

```
npx create-react-app client
```

You should now have a folder named "client" like so



Step 2: Test your app

```
cd client
npm start
```

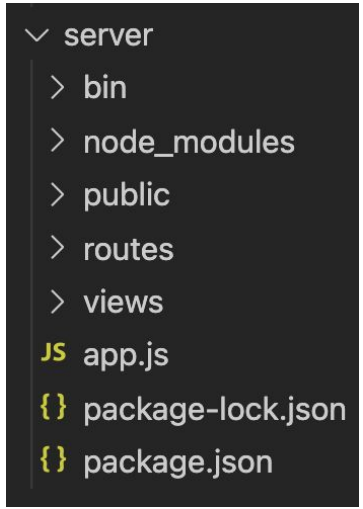
You should be able to view the react homepage on your browser at http://localhost:3000

Running React On Node: Installing Express

Step 3: Installing Express

```
npx express-generator server
```

This creates an express application on NodeJS named server



Step 4: Inspect package.json

```
npx install
```

This installs all the dependencies mentioned in this file
Test your installation using :

```
cd server
npm start
```

Running React On Node: Adding Routes

Step 5: Change port number to say 4000 in server/bin/www

Step 6: Create a new file api.js in server/routes

server/routes/api.js

```
var express = require('express');
var router = express.Router();


router.get('/', function(req, res, next) {
 res.send("You have successfully connected a React App to NodeJS!");
});

module.exports = router;
```

Running React On Node: Creating Your API

Step 7: In server/app.js, import the newly created module

```
var apiRouter=require("./routes/api");

app.use('/api', apiRouter );
```

Now test using npm start
Check http://localhost:4000/api
You should be able to view the message.

Step 8: Create your react component and export in client/src/App.js as shown in the next slide.

client/src/App.js

```
import React from 'react';
import './App.css';
class App extends React.Component{
 constructor(){
   super();
   this.state={response:""};
 }
 fetchResponse(){
   fetch("http://localhost:9000/api")
   .then(res => res.text())
   .then(res => this.setState({response:res}));
 }
 componentWillMount(){
   this.fetchResponse();
 }
 render() {
   return (
     <div className="App">
       <header className="App-header">
         <p>
           {this.state.response}
         </p>
       </header>
     </div>
   );
 }
}

export default App;
```

componentWillMount is executed before rendering! Think about what would happen if you use componentDidMount

Running React On Node: Handling CORS

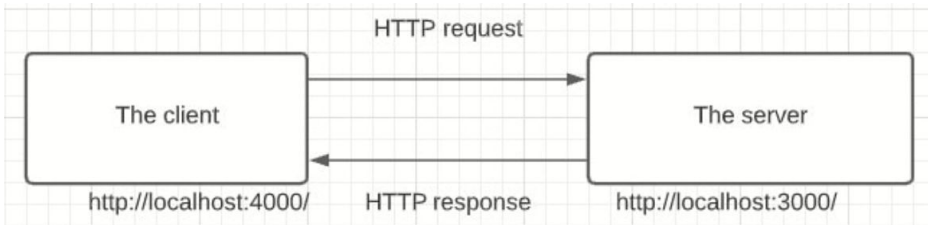
Step 9: The client and server have a different origin from each other, trying to make a request to a resource on the other server will fail. Therefore, we need to install a module to enable cross origin resource sharing in server/app.js

```
npm install --save cors
```

Import this module in server/app.js like so:

```
var cors = require('cors');
app.use(cors());
```

Take care to add in this statement in the lines before adding your api!

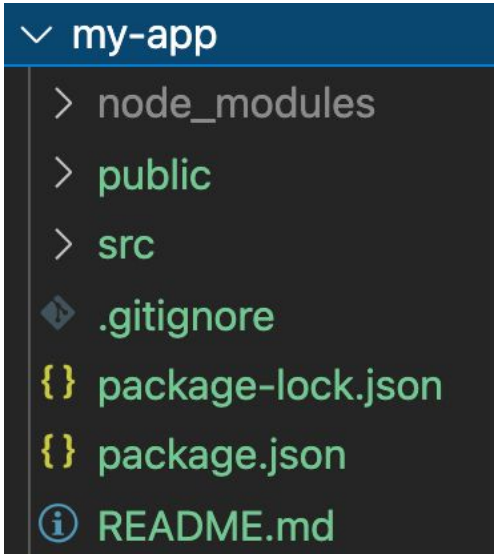


Step 10: Run client and server using npm start and check localhost:3000 to see your integrated application!

React Router

Create React App is a comfortable environment for learning React, and is the best way to start building a new single-page application in React.

```
npx create-react-app my-app
cd my-app
npm start
```



However, Create React App doesn't include page routing.

This is where React Router comes into the picture.
It is a lightweight, fully-featured routing library for the React JavaScript library.

React Router

Many modern websites are actually made up of a single page, they just look like multiple pages because they contain components which render like separate pages.

These are usually referred to as SPAs - **s**ingle-**p**age **a**pplications.

At its core, what React Router does is conditionally render certain components to display depending on the *route* being used in the URL (/ for the home page, /about for the about page, etc.).

To use React Router, you first have to install it using NPM:

```
npm install react-router-dom
```

React Router Components

- **<BrowserRouter></BrowserRouter>**
  It is used to wrap different routes

- **<Routes></Routes>**
  Renders the first child <Route> that matches the location.

- **<Route path='path-to-be-matched' element={<component-to-be-rendered/>}**
  It is responsible for rendering the UI of a React component. It has a prop called path which always matches the current URL of the application. The second required prop is called element that tells the Route component when a current URL is encountered and which React component to be rendered.

- **<Link to='path'></Link>**
  The primary way to allow users to navigate around your application. It takes a prop 'to' which can contain a path

React Router

Let's dive into routing with react router. First within the src folder, create 3 files : home.js, about.js and contact.js

home.js

```
import React from 'react';
class Home extends React.Component {
   render() {
      return (
         <div>
            <h1>Home</h1>
         </div>
      )
   }
}
export default Home;
```

about.js

```
import React from 'react';
class About extends React.Component {
   render() {
      return (
         <div>
            <h1>About...</h1>
         </div>
      )
   }
}
export default About;
```

React Router

contact.js

```
import React from 'react';
class Contact extends React.Component {
   render() {
      return (
         <div>
            <h1>Contact...</h1>
         </div>
      )
   }
}
export default Contact;
```

React Router

```
import ReactDOM from "react-dom";
import { BrowserRouter as Router, Routes, Route, Link } from "react-router-dom";
import Home from "./home";
import Contact from "./contact";
import About from "./about";
export default function App() {
 return (
   <Router>
     <div><Link to="/">Home</Link></div>
     <div><Link to="/about">about </Link></div>
     <div><Link to="/contact">Contact Me</Link></div>
     <hr />
     <Routes>
       <Route path="/" element={<Home/>}/>
       <Route path="/about" element={<About/>}/>
       <Route path="/contact" element={<Contact/>}/>
     </Routes>
   </Router>
 );}
ReactDOM.render(<App />, document.getElementById("root"));
```

Import components

<Routes/> ensures that only one of the routes is active at any given point of time!

THANK YOU

Prof. Spurthi N Anjan
Department of Computer Science and Engineering
spurthianjan@pes.edu

Sinduja Mullangi
Teaching Assistant
sinduja.mullangi@gmail.com